PHASE1-1.41 PART1: Vultr Cost Metrics Collector - Code Implementation

# PHASE1-1.41 PART1: Vultr Cost Metrics Collector - Code Implementation

**Document Version:** 1.0
**Date:** October 21, 2025
**Phase:** Cost Agent - Week 2
**Status:** Ready for Implementation
**Prerequisites:** PHASE1-1.2 (AWS Collector) completed

---

## 📋 TABLE OF CONTENTS

---

## 🎯 OVERVIEW

### Purpose

This phase implements the Vultr Cost Metrics Collector, enabling OptiInfra to: - **Collect billing data** from Vultr AI Cloud infrastructure - **Track compute, GPU, and storage costs** across all Vultr services - **Analyze spending patterns** for optimization opportunities - **Identify cost-saving opportunities** specific to Vultr's AI/GPU offerings

### Vultr Context

Vultr is an AI-focused cloud provider with 32 cities across 19 countries, offering Cloud Compute, Cloud GPU, Bare Metal, and AI/ML services with competitive hourly billing.

### Time Estimate

- **Code Generation:** 25 minutes
- **Verification:** 20 minutes
- **Total:** ~45 minutes

### Success Criteria

✅ Vultr API integration working
✅ Billing data collection successful
✅ Cost metrics stored in ClickHouse
✅ Instance and GPU tracking functional
✅ Tests pass with 80%+ coverage

---

## 🏗 WHAT YOU'LL BUILD

### Core Components

1. **Vultr Client** (`src/collectors/vultr/client.py`)
   - API authentication with Bearer token
   - Rate limiting (30 calls/second limit)
   - Error handling and retries
   - Pagination support
2. **Billing Collector** (`src/collectors/vultr/billing.py`)
   - Invoice retrieval and parsing
   - Pending charges collection
   - Account balance tracking
   - Cost breakdown by service
3. **Instance Collector** (`src/collectors/vultr/instances.py`)
   - Cloud Compute instances
   - Cloud GPU instances
   - Bare Metal servers
   - Resource tagging and metadata
4. **Cost Analyzer** (`src/collectors/vultr/analyzer.py`)
   - Cost aggregation by service type
   - Usage pattern analysis
   - GPU vs CPU cost breakdown
   - Optimization recommendations

---

## 🌐 VULTR API OVERVIEW

### API Characteristics

**Base URL:** `https://api.vultr.com/v2`

**Authentication:** Bearer token in Authorization header

**Rate Limits:** 30 calls per second (default: 500ms per call, 3 retries)

**Key Endpoints:**

```
GET /account                        # Account info & balance
GET /billing/history                # Billing history
GET /billing/invoices               # List invoices
GET /billing/invoices/{id}          # Invoice details
GET /billing/invoices/{id}/items      # Invoice line items
GET /billing/pending-charges          # Current month charges
GET /instances                        # Cloud Compute instances
GET /bare-metals                      # Bare Metal servers
GET /plans                            # Available plans/pricing
```

### Billing Model

Vultr uses different billing models: Cloud VMs use 28-day (672-hour) months, while GPU products use 730-hour months for hourly billing.

### Cost Structure

Common charges include: - Compute: Hourly instance costs - Storage: $0.05/GB per month for snapshots - DDoS Protection: $10/month per instance (10Gbps mitigation) - Backups: 20% of base instance cost

---

## 🏛 ARCHITECTURE

### Data Flow

```
┌─────────────────────────────────────────────────────────┐
│                  Vultr Cost Collector                     │
├─────────────────────────────────────────────────────────┤
│                                                           │
│  ┌──────────────┐      ┌──────────────┐                 │
│  │   Account    │      │   Billing    │                 │
│  │   Client     │─────▶│  Collector   │                 │
│  └──────────────┘      └──────────────┘                 │
│         │                      │                          │
│         │                      ▼                          │
│         │              ┌──────────────┐                 │
│         │              │   Invoice    │                 │
│         │              │   Parser     │                 │
│         │              └──────────────┘                 │
│         │                      │                          │
│         ▼                      │                          │
│  ┌──────────────┐             │                         │
│  │  Instance    │             │                         │
│  │  Collector   │             │                         │
│  └──────────────┘             │                         │
│         │                      │                          │
│         │             ┌────────┘                          │
│         │             │                                   │
│         ▼             ▼                                   │
│  ┌──────────────────────────────┐                       │
│  │     Cost Analyzer             │                        │
│  │  - Aggregate by service       │                        │
│  │  - GPU vs CPU breakdown       │                        │
│  │  - Identify waste             │                        │
│  └──────────────────────────────┘                       │
│                 │                                         │
│                 ▼                                         │
│  ┌──────────────────────────────┐                       │
│  │      ClickHouse Storage       │                        │
│  │  - cost_metrics table         │                        │
│  │  - instance_metadata          │                        │
│  └──────────────────────────────┘                       │
│                                                           │
└─────────────────────────────────────────────────────────┘
```

### Cost Metrics Schema

```
{
    "customer_id": "cust_123",
    "cloud_provider": "vultr",
    "timestamp": "2025-10-21T00:00:00Z",
    "billing_period": "2025-10",

    # Account-level metrics
    "account_balance": 150.50,
    "pending_charges": 45.30,
    "monthly_spend": 320.00,

    # Service breakdown
    "compute_cost": 180.00,
    "gpu_cost": 90.00,
    "storage_cost": 25.00,
    "network_cost": 15.00,
    "backup_cost": 10.00,

    # Instance details
    "instance_count": 15,
    "gpu_instance_count": 3,
    "bare_metal_count": 2,

    # Optimization metrics
    "idle_instances": 2,
    "underutilized_resources": 5,
    "snapshot_waste": 50.00
}
```

---

## 📦 DEPENDENCIES

### Python Packages

Add to `requirements.txt`:

```
# Existing dependencies
fastapi==0.104.1
uvicorn[standard]==0.24.0
pydantic==2.5.0
sqlalchemy==2.0.23
asyncpg==0.29.0
redis==5.0.1
boto3==1.29.7  # AWS
google-cloud-billing==1.11.5  # GCP
azure-mgmt-costmanagement==4.0.1  # Azure

# NEW: Vultr dependencies
requests==2.31.0
aiohttp==3.9.1
tenacity==8.2.3  # For retry logic

# Testing
pytest==7.4.3
pytest-asyncio==0.21.1
pytest-cov==4.1.0
responses==0.24.1  # Mock HTTP requests
```

### Installation

```
cd services/cost-agent
pip install requests aiohttp tenacity responses --break-system-packages
```

---

## 🔧 IMPLEMENTATION STEPS

### Step 1: Create Vultr Client

**File:** `services/cost-agent/src/collectors/vultr/client.py`

```
"""
Vultr API client with authentication, rate limiting, and error handling.
"""

import os
import time
```

```python
from typing import Dict, Any, Optional, List
from datetime import datetime
import logging

import requests
from tenacity import (
    retry,
    stop_after_attempt,
    wait_exponential,
    retry_if_exception_type
)

logger = logging.getLogger(__name__)


class VultrAPIError(Exception):
    """Base exception for Vultr API errors"""
    pass


class VultrRateLimitError(VultrAPIError):
    """Raised when rate limit is exceeded"""
    pass


class VultrAuthenticationError(VultrAPIError):
    """Raised when authentication fails"""
    pass


class VultrClient:
    """
    Client for Vultr API v2.
    Handles authentication, rate limiting, and common API
operations.
    """

    BASE_URL = "https://api.vultr.com/v2"

    # Rate limiting: 30 calls per second = 1 call per 33ms
    # We'll be conservative: 500ms per call (default)
    RATE_LIMIT_DELAY = 0.5  # seconds

    def __init__(
        self,
        api_key: Optional[str] = None,
        rate_limit_delay: float = RATE_LIMIT_DELAY
    ):
        """
        Initialize Vultr API client.

        Args:
            api_key: Vultr API key (or set VULTR_API_KEY env var)
            rate_limit_delay: Delay between API calls in seconds
        """
        self.api_key = api_key or os.getenv("VULTR_API_KEY")
        if not self.api_key:
            raise VultrAuthenticationError(
                "Vultr API key required. Set VULTR_API_KEY
environment variable."
            )

        self.rate_limit_delay = rate_limit_delay
        self.last_request_time = 0

        # Setup session
        self.session = requests.Session()
        self.session.headers.update({
            "Authorization": f"Bearer {self.api_key}",
            "Content-Type": "application/json"
        })

        logger.info("Vultr client initialized")

    def _wait_for_rate_limit(self):
```

```python
                """Wait to respect rate limiting"""
                elapsed = time.time() - self.last_request_time
                if elapsed < self.rate_limit_delay:
                    time.sleep(self.rate_limit_delay - elapsed)
                self.last_request_time = time.time()

            @retry(
                stop=stop_after_attempt(3),
                wait=wait_exponential(multiplier=1, min=2, max=10),
                retry=retry_if_exception_type((requests.RequestException,
VultrRateLimitError)),
                reraise=True
            )
            def _request(
                self,
                method: str,
                endpoint: str,
                params: Optional[Dict[str, Any]] = None,
                data: Optional[Dict[str, Any]] = None
            ) -> Dict[str, Any]:
                """
                Make an API request with rate limiting and retries.

                Args:
                    method: HTTP method (GET, POST, etc.)
                    endpoint: API endpoint (without base URL)
                    params: Query parameters
                    data: Request body data

                Returns:
                    Response JSON data

                Raises:
                    VultrAPIError: On API errors
                    VultrRateLimitError: On rate limit exceeded
                    VultrAuthenticationError: On authentication failure
                """
                self._wait_for_rate_limit()

                url = f"{self.BASE_URL}/{endpoint.lstrip('/')}"

                try:
                    response = self.session.request(
                        method=method,
                        url=url,
                        params=params,
                        json=data,
                        timeout=30
                    )

                    # Handle rate limiting
                    if response.status_code == 429:
                        logger.warning("Rate limit exceeded, will retry")
                        raise VultrRateLimitError("Rate limit exceeded")

                    # Handle authentication errors
                    if response.status_code == 401:
                        raise VultrAuthenticationError("Invalid API key")

                    # Handle other errors
                    if response.status_code >= 400:
                        error_msg = response.json().get("error",
response.text)
                        raise VultrAPIError(
                            f"API error {response.status_code}: {error_msg}"
                        )

                    return response.json()

                except requests.RequestException as e:
                    logger.error(f"Request failed: {e}")
                    raise

            def get(
                self,
                endpoint: str,
                params: Optional[Dict[str, Any]] = None
            ) -> Dict[str, Any]:
                """Make a GET request"""
                return self._request("GET", endpoint, params=params)

            def post(
                self,
                endpoint: str,
                data: Dict[str, Any]
            ) -> Dict[str, Any]:
                """Make a POST request"""
                return self._request("POST", endpoint, data=data)

            def get_paginated(
                self,
                endpoint: str,
                params: Optional[Dict[str, Any]] = None,
                per_page: int = 100
            ) -> List[Dict[str, Any]]:
                """
                Get all pages of results from a paginated endpoint.

                Args:
                    endpoint: API endpoint
                    params: Query parameters
                    per_page: Items per page (max 500)

                Returns:
                    List of all items across all pages
                """
                all_items = []
                params = params or {}
                params["per_page"] = min(per_page, 500)  # Vultr max is 500

                cursor = None

                while True:
                    if cursor:
                        params["cursor"] = cursor

                    response = self.get(endpoint, params=params)

                    # Extract items (key varies by endpoint)
                    # Common keys: instances, invoices, bare_metals, etc.
                    items = None
                    for key in response.keys():
                        if isinstance(response[key], list):
                            items = response[key]
                            break

                    if items:
                        all_items.extend(items)

                    # Check for next page
                    meta = response.get("meta", {})
                    links = meta.get("links", {})
                    cursor = links.get("next")

                    if not cursor:
                        break

                    logger.debug(f"Fetching next page: {cursor}")

                logger.info(f"Retrieved {len(all_items)} items from
{endpoint}")
                return all_items

            # ============================================================
            # CONVENIENCE METHODS
            # ============================================================

            def get_account_info(self) -> Dict[str, Any]:
                """Get account information including balance"""
                return self.get("/account")
```

```
    def list_invoices(self) -> List[Dict[str, Any]]:
        """List all invoices"""
        return self.get_paginated("/billing/invoices")

    def get_invoice(self, invoice_id: str) -> Dict[str, Any]:
        """Get specific invoice details"""
        return self.get(f"/billing/invoices/{invoice_id}")

    def get_invoice_items(self, invoice_id: str) -> List[Dict[str,
Any]]:
        """Get line items for an invoice"""
        response = self.get(f"/billing/invoices/{invoice_id}/items")
        return response.get("invoice_items", [])

    def get_pending_charges(self) -> Dict[str, Any]:
        """Get current month's pending charges"""
        return self.get("/billing/pending-charges")

    def get_billing_history(self) -> Dict[str, Any]:
        """Get billing history"""
        return self.get("/billing/history")

    def list_instances(self) -> List[Dict[str, Any]]:
        """List all Cloud Compute instances"""
        return self.get_paginated("/instances")

    def list_bare_metals(self) -> List[Dict[str, Any]]:
        """List all Bare Metal servers"""
        return self.get_paginated("/bare-metals")

    def list_plans(self) -> List[Dict[str, Any]]:
        """List all available plans"""
        return self.get_paginated("/plans")


# Async version for async contexts
class AsyncVultrClient(VultrClient):
    """
    Async version of Vultr client using aiohttp.
    Use this in async/await contexts.
    """

    def __init__(self, *args, **kwargs):
        import aiohttp
        super().__init__(*args, **kwargs)
        self.async_session = None

    async def __aenter__(self):
        import aiohttp
        self.async_session = aiohttp.ClientSession(
            headers={
                "Authorization": f"Bearer {self.api_key}",
                "Content-Type": "application/json"
            }
        )
        return self

    async def __aexit__(self, exc_type, exc_val, exc_tb):
        if self.async_session:
            await self.async_session.close()

    async def _async_request(
        self,
        method: str,
        endpoint: str,
        params: Optional[Dict] = None,
        data: Optional[Dict] = None
    ) -> Dict[str, Any]:
        """Async version of _request"""
        import aiohttp

        self._wait_for_rate_limit()

        url = f"{self.BASE_URL}/{endpoint.lstrip('/')}"
```

```
                async with self.async_session.request(
                    method=method,
                    url=url,
                    params=params,
                    json=data,
                    timeout=aiohttp.ClientTimeout(total=30)
                ) as response:

                    if response.status == 429:
                        raise VultrRateLimitError("Rate limit exceeded")

                    if response.status == 401:
                        raise VultrAuthenticationError("Invalid API key")

                    if response.status >= 400:
                        error_text = await response.text()
                        raise VultrAPIError(f"API error {response.status}:
{error_text}")

                    return await response.json()

            async def get_async(
                self,
                endpoint: str,
                params: Optional[Dict] = None
            ) -> Dict[str, Any]:
                """Async GET request"""
                return await self._async_request("GET", endpoint,
params=params)
```

## Step 2: Create Billing Collector

**File:** `services/cost-agent/src/collectors/vultr/billing.py`

```
"""
Vultr billing data collector.
Collects invoices, pending charges, and cost breakdowns.
"""

from typing import Dict, Any, List, Optional
from datetime import datetime, timedelta
import logging

from .client import VultrClient

logger = logging.getLogger(__name__)


class VultrBillingCollector:
    """
    Collects billing data from Vultr API.
    """

    def __init__(self, client: VultrClient):
        """
        Initialize billing collector.

        Args:
            client: Configured VultrClient instance
        """
        self.client = client

    def collect_account_info(self) -> Dict[str, Any]:
        """
        Collect account information including balance.

        Returns:
            Account information dict
        """
        try:
            account = self.client.get_account_info()

            return {
```

```
                        "account_id": account.get("account", {}).get("name",
"unknown"),
                        "email": account.get("account", {}).get("email"),
                        "balance": float(account.get("account",
{}).get("balance", 0)),
                        "pending_charges": float(
                            account.get("account",
{}).get("pending_charges", 0)
                        ),
                        "last_payment_date": account.get("account", {}).get(
                            "last_payment_date"
                        ),
                        "last_payment_amount": float(
                            account.get("account",
{}).get("last_payment_amount", 0)
                        ),
                    }

                except Exception as e:
                    logger.error(f"Failed to collect account info: {e}")
                    raise

            def collect_pending_charges(self) -> Dict[str, Any]:
                """
                Collect current month's pending charges.

                Returns:
                    Pending charges breakdown
                """
                try:
                    charges = self.client.get_pending_charges()

                    billing_info = charges.get("billing", {})

                    return {
                        "pending_charges": float(
                            billing_info.get("pending_charges", 0)
                        ),
                        "currency": "USD",  # Vultr bills in USD
                        "billing_period_start":
billing_info.get("billing_period_start"),
                        "billing_period_end":
billing_info.get("billing_period_end"),
                    }

                except Exception as e:
                    logger.error(f"Failed to collect pending charges: {e}")
                    raise

            def collect_invoices(
                self,
                start_date: Optional[datetime] = None,
                end_date: Optional[datetime] = None
            ) -> List[Dict[str, Any]]:
                """
                Collect invoices within date range.

                Args:
                    start_date: Start date (default: 90 days ago)
                    end_date: End date (default: now)

                Returns:
                    List of invoice summaries
                """
                if not start_date:
                    start_date = datetime.utcnow() - timedelta(days=90)
                if not end_date:
                    end_date = datetime.utcnow()

                try:
                    invoices = self.client.list_invoices()

                    # Filter by date
                    filtered_invoices = []
                    for invoice in invoices:
```

```python
                invoice_date = datetime.fromisoformat(
                    invoice.get("date", "").replace("Z", "+00:00")
                )
                if start_date <= invoice_date <= end_date:
                    filtered_invoices.append({
                        "invoice_id": invoice.get("id"),
                        "date": invoice.get("date"),
                        "description": invoice.get("description"),
                        "amount": float(invoice.get("amount", 0)),
                        "balance": float(invoice.get("balance", 0)),
                    })

            logger.info(
                f"Collected {len(filtered_invoices)} invoices "
                f"from {start_date} to {end_date}"
            )
            return filtered_invoices

        except Exception as e:
            logger.error(f"Failed to collect invoices: {e}")
            raise

    def collect_invoice_details(
        self,
        invoice_id: str
    ) -> Dict[str, Any]:
        """
        Collect detailed line items for a specific invoice.

        Args:
            invoice_id: Invoice ID

        Returns:
            Invoice details with line items
        """
        try:
            # Get invoice overview
            invoice = self.client.get_invoice(invoice_id)

            # Get line items
            items = self.client.get_invoice_items(invoice_id)

            # Group by product type
            product_costs = {}
            for item in items:
                product = item.get("product", "unknown")
                amount = float(item.get("total", 0))

                if product not in product_costs:
                    product_costs[product] = {
                        "product": product,
                        "total_cost": 0,
                        "items": []
                    }

                product_costs[product]["total_cost"] += amount
                product_costs[product]["items"].append({
                    "description": item.get("description"),
                    "start_date": item.get("start_date"),
                    "end_date": item.get("end_date"),
                    "units": item.get("units"),
                    "unit_type": item.get("unit_type"),
                    "unit_price": float(item.get("unit_price", 0)),
                    "total": amount
                })

            return {
                "invoice_id": invoice_id,
                "date": invoice.get("invoice", {}).get("date"),
                "total_amount": float(
                    invoice.get("invoice", {}).get("amount", 0)
                ),
                "product_breakdown": list(product_costs.values()),
            }
```

```
                except Exception as e:
                    logger.error(f"Failed to collect invoice details: {e}")
                    raise

            def analyze_spending_patterns(
                self,
                invoices: List[Dict[str, Any]]
            ) -> Dict[str, Any]:
                """
                Analyze spending patterns from invoices.

                Args:
                    invoices: List of invoice summaries

                Returns:
                    Spending analysis
                """
                if not invoices:
                    return {
                        "total_spend": 0,
                        "average_monthly": 0,
                        "trend": "unknown"
                    }

                # Calculate totals
                total_spend = sum(inv["amount"] for inv in invoices)

                # Calculate monthly average
                date_range_days = (
                    datetime.fromisoformat(invoices[0]["date"].replace("Z",
"+00:00")) -

                    datetime.fromisoformat(invoices[-1]["date"].replace("Z",
"+00:00"))
                ).days
                months = max(date_range_days / 30, 1)
                average_monthly = total_spend / months

                # Determine trend (simple: compare first half vs second
half)
                mid_point = len(invoices) // 2
                first_half_avg = sum(
                    inv["amount"] for inv in invoices[:mid_point]
                ) / max(mid_point, 1)
                second_half_avg = sum(
                    inv["amount"] for inv in invoices[mid_point:]
                ) / max(len(invoices) - mid_point, 1)

                if second_half_avg > first_half_avg * 1.1:
                    trend = "increasing"
                elif second_half_avg < first_half_avg * 0.9:
                    trend = "decreasing"
                else:
                    trend = "stable"

                return {
                    "total_spend": total_spend,
                    "average_monthly": average_monthly,
                    "invoice_count": len(invoices),
                    "trend": trend,
                    "trend_percentage": (
                        (second_half_avg - first_half_avg) / first_half_avg
* 100
                        if first_half_avg > 0 else 0
                    )
                }
```

### Step 3: Create Instance Collector

**File:** `services/cost-agent/src/collectors/vultr/instances.py`

```
"""
Vultr instance collector.
Collects compute, GPU, and bare metal instance data.
"""
```

```
        from typing import Dict, Any, List
        import logging

        from .client import VultrClient

        logger = logging.getLogger(__name__)


        class VultrInstanceCollector:
            """
            Collects instance data from Vultr API.
            """

            def __init__(self, client: VultrClient):
                """
                Initialize instance collector.

                Args:
                    client: Configured VultrClient instance
                """
                self.client = client

            def collect_compute_instances(self) -> List[Dict[str, Any]]:
                """
                Collect all Cloud Compute instances.

                Returns:
                    List of instance details
                """
                try:
                    instances = self.client.list_instances()

                    collected_instances = []
                    for instance in instances:
                        # Determine if GPU instance
                        plan = instance.get("plan", "")
                        is_gpu = "gpu" in plan.lower() or "vhf" in
plan.lower()

                        collected_instances.append({
                            "instance_id": instance.get("id"),
                            "label": instance.get("label"),
                            "hostname": instance.get("hostname"),
                            "plan": plan,
                            "region": instance.get("region"),
                            "os": instance.get("os"),
                            "status": instance.get("status"),
                            "power_status": instance.get("power_status"),
                            "vcpu_count": instance.get("vcpu_count"),
                            "ram": instance.get("ram"),  # in MB
                            "disk": instance.get("disk"),  # in GB
                            "is_gpu": is_gpu,
                            "monthly_cost":
float(instance.get("monthly_cost", 0)),
                            "tags": instance.get("tags", []),
                            "created_at": instance.get("date_created"),
                        })

                    logger.info(f"Collected {len(collected_instances)}
compute instances")
                    return collected_instances

                except Exception as e:
                    logger.error(f"Failed to collect compute instances:
{e}")
                    raise

            def collect_bare_metal_servers(self) -> List[Dict[str, Any]]:
                """
                Collect all Bare Metal servers.

                Returns:
                    List of bare metal server details
                """
                try:
                    servers = self.client.list_bare_metals()

                    collected_servers = []
                    for server in servers:
                        collected_servers.append({
                            "server_id": server.get("id"),
                            "label": server.get("label"),
                            "plan": server.get("plan"),
                            "region": server.get("region"),
                            "os": server.get("os"),
                            "status": server.get("status"),
                            "cpu_count": server.get("cpu_count"),
                            "ram": server.get("ram"),
                            "disk": server.get("disk"),
                            "monthly_cost": float(server.get("monthly_cost",
0)),

                            "tags": server.get("tags", []),
                            "created_at": server.get("date_created"),
                        })

                    logger.info(f"Collected {len(collected_servers)} bare
metal servers")
                    return collected_servers

                except Exception as e:
                    logger.error(f"Failed to collect bare metal servers:
{e}")
                    raise

            def analyze_instance_utilization(
                self,
                instances: List[Dict[str, Any]]
            ) -> Dict[str, Any]:
                """
                Analyze instance utilization patterns.

                Args:
                    instances: List of instances

                Returns:
                    Utilization analysis
                """
                total_instances = len(instances)

                # Count by status
                running = sum(1 for i in instances if i["status"] ==
"active")
                stopped = sum(1 for i in instances if i["status"] !=
"active")

                # GPU instances
                gpu_instances = [i for i in instances if i.get("is_gpu")]
                gpu_count = len(gpu_instances)

                # Calculate costs
                total_monthly_cost = sum(i["monthly_cost"] for i in
instances)
                gpu_monthly_cost = sum(i["monthly_cost"] for i in
gpu_instances)

                # Identify idle (stopped but still charged)
                idle_instances = [
                    i for i in instances
                    if i["power_status"] == "stopped" and i["monthly_cost"]
> 0
                ]
                idle_cost = sum(i["monthly_cost"] for i in idle_instances)

                return {
                    "total_instances": total_instances,
                    "running_instances": running,
                    "stopped_instances": stopped,
                    "gpu_instances": gpu_count,
                    "gpu_percentage": (gpu_count / total_instances * 100) if
total_instances else 0,
                    "total_monthly_cost": total_monthly_cost,
                    "gpu_monthly_cost": gpu_monthly_cost,
                    "idle_instances": len(idle_instances),
                    "idle_cost": idle_cost,
                    "idle_waste_percentage": (
                        (idle_cost / total_monthly_cost * 100) if
total_monthly_cost else 0
                    )
                }
```

### Step 4: Create Cost Analyzer

**File:** `services/cost-agent/src/collectors/vultr/analyzer.py`

```
        """
        Vultr cost analyzer.
        Analyzes costs and identifies optimization opportunities.
        """

        from typing import Dict, Any, List
        from datetime import datetime
        import logging

        logger = logging.getLogger(__name__)


        class VultrCostAnalyzer:
            """
            Analyzes Vultr costs and identifies savings opportunities.
            """

            def analyze_costs(
                self,
                account_info: Dict[str, Any],
                pending_charges: Dict[str, Any],
                instances: List[Dict[str, Any]],
                invoices: List[Dict[str, Any]]
            ) -> Dict[str, Any]:
                """
                Comprehensive cost analysis.

                Args:
                    account_info: Account information
                    pending_charges: Current month charges
                    instances: List of instances
                    invoices: Historical invoices

                Returns:
                    Cost analysis with recommendations
                """
                # Calculate current spend
                current_monthly = pending_charges.get("pending_charges", 0)

                # GPU vs CPU breakdown
                gpu_instances = [i for i in instances if i.get("is_gpu")]
                cpu_instances = [i for i in instances if not
i.get("is_gpu")]

                gpu_cost = sum(i.get("monthly_cost", 0) for i in
gpu_instances)
                cpu_cost = sum(i.get("monthly_cost", 0) for i in
cpu_instances)

                # Idle resources
                idle_instances = [
                    i for i in instances
                    if i.get("power_status") == "stopped" and
i.get("monthly_cost", 0) > 0
                ]
                idle_cost = sum(i.get("monthly_cost", 0) for i in
idle_instances)

                # Recommendations
                recommendations = []
                estimated_savings = 0

                # Recommendation 1: Delete idle instances
                if idle_instances:
                    recommendations.append({
                        "type": "delete_idle_instances",
                        "priority": "high",
                        "description": f"Delete {len(idle_instances)}
stopped instances",
                        "instances": [i["instance_id"] for i in
idle_instances],
                        "estimated_savings": idle_cost,
                        "confidence": 0.95
                    })
                    estimated_savings += idle_cost

                # Recommendation 2: Snapshot optimization
                # Vultr charges $0.05/GB for snapshots
                # This would require additional API calls to get snapshot
data

                # Recommendation 3: Right-size underutilized instances
                # Would require utilization metrics (CPU/RAM usage)
                # Placeholder for now

                return {
                    "timestamp": datetime.utcnow().isoformat(),
                    "cloud_provider": "vultr",

                    # Current state
                    "account_balance": account_info.get("balance", 0),
                    "current_monthly_spend": current_monthly,
                    "instance_count": len(instances),

                    # Breakdown
                    "cost_breakdown": {
                        "gpu_cost": gpu_cost,
                        "cpu_cost": cpu_cost,
                        "gpu_percentage": (
                            (gpu_cost / (gpu_cost + cpu_cost) * 100)
                            if (gpu_cost + cpu_cost) > 0 else 0
                        ),
                    },

                    # Waste identification
                    "waste_analysis": {
                        "idle_instances": len(idle_instances),
                        "idle_cost": idle_cost,
                        "idle_percentage": (
                            (idle_cost / current_monthly * 100)
                            if current_monthly > 0 else 0
                        ),
                    },

                    # Recommendations
                    "recommendations": recommendations,
                    "total_estimated_savings": estimated_savings,
                    "savings_percentage": (
                        (estimated_savings / current_monthly * 100)
                        if current_monthly > 0 else 0
                    ),
                }

            def compare_with_competitors(
                self,
                vultr_costs: Dict[str, Any]
            ) -> Dict[str, Any]:
                """
                Compare Vultr costs with AWS/GCP/Azure.

                Args:
                    vultr_costs: Current Vultr cost analysis

                Returns:
```

```
            Comparison analysis
        """
        # This is a simplified comparison
        # In reality, would need to compare similar instance types

        # Vultr is generally 20-40% cheaper than AWS/Azure
        # for comparable instances

        current_spend = vultr_costs["current_monthly_spend"]

        return {
            "vultr_cost": current_spend,
            "estimated_aws_cost": current_spend * 1.3,  # 30% more
            "estimated_gcp_cost": current_spend * 1.25,  # 25% more
            "estimated_azure_cost": current_spend * 1.35,  # 35%
more
            "vultr_savings_vs_aws": current_spend * 0.3,
            "vultr_savings_vs_gcp": current_spend * 0.25,
            "vultr_savings_vs_azure": current_spend * 0.35,
            "note": "Estimates based on typical instance pricing
comparisons"
        }
```

### Step 5: Create Main Collector

**File:** `services/cost-agent/src/collectors/vultr/__init__.py`

```python
"""
Vultr cost collector module.
"""

from .client import VultrClient, AsyncVultrClient, VultrAPIError
from .billing import VultrBillingCollector
from .instances import VultrInstanceCollector
from .analyzer import VultrCostAnalyzer

__all__ = [
    "VultrClient",
    "AsyncVultrClient",
    "VultrAPIError",
    "VultrBillingCollector",
    "VultrInstanceCollector",
    "VultrCostAnalyzer",
    "collect_vultr_metrics",
]


def collect_vultr_metrics(api_key: str) -> dict:
    """
    Convenience function to collect all Vultr metrics.

    Args:
        api_key: Vultr API key

    Returns:
        Complete cost metrics
    """
    # Initialize client
    client = VultrClient(api_key=api_key)

    # Initialize collectors
    billing_collector = VultrBillingCollector(client)
    instance_collector = VultrInstanceCollector(client)
    analyzer = VultrCostAnalyzer()

    # Collect data
    account_info = billing_collector.collect_account_info()
    pending_charges = billing_collector.collect_pending_charges()
    instances = instance_collector.collect_compute_instances()
    bare_metals = instance_collector.collect_bare_metal_servers()
    invoices = billing_collector.collect_invoices()

    # Combine instances
    all_instances = instances + bare_metals
```

```
        # Analyze
        cost_analysis = analyzer.analyze_costs(
            account_info=account_info,
            pending_charges=pending_charges,
            instances=all_instances,
            invoices=invoices
        )

        return {
            "account": account_info,
            "pending_charges": pending_charges,
            "instances": all_instances,
            "cost_analysis": cost_analysis,
            "collected_at": cost_analysis["timestamp"]
        }
```

## Step 6: Create Tests

**File:** `services/cost-agent/tests/test_vultr_collector.py`

```
"""
Tests for Vultr cost collector.
"""

import pytest
from unittest.mock import Mock, patch
import responses

from src.collectors.vultr import (
    VultrClient,
    VultrBillingCollector,
    VultrInstanceCollector,
    VultrCostAnalyzer,
    VultrAPIError,
    collect_vultr_metrics
)


class TestVultrClient:
    """Test Vultr API client"""

    @responses.activate
    def test_authentication(self):
        """Test API authentication"""
        responses.add(
            responses.GET,
            "https://api.vultr.com/v2/account",
            json={"account": {"name": "test", "balance": 100}},
            status=200
        )

        client = VultrClient(api_key="test_key")
        result = client.get_account_info()

        assert result["account"]["name"] == "test"
        assert responses.calls[0].request.headers["Authorization"]
== "Bearer test_key"

    @responses.activate
    def test_rate_limiting(self):
        """Test rate limiting works"""
        import time

        responses.add(
            responses.GET,
            "https://api.vultr.com/v2/account",
            json={"account": {}},
            status=200
        )

        client = VultrClient(api_key="test_key",
rate_limit_delay=0.1)
```

```python
        start = time.time()
        client.get_account_info()
        client.get_account_info()
        elapsed = time.time() - start

        # Should take at least 0.1 seconds due to rate limiting
        assert elapsed >= 0.1

    @responses.activate
    def test_pagination(self):
        """Test paginated requests"""
        # First page
        responses.add(
            responses.GET,
            "https://api.vultr.com/v2/instances",
            json={
                "instances": [{"id": "1"}, {"id": "2"}],
                "meta": {"links": {"next": "cursor123"}}
            },
            status=200
        )

        # Second page
        responses.add(
            responses.GET,
            "https://api.vultr.com/v2/instances",
            json={
                "instances": [{"id": "3"}],
                "meta": {"links": {"next": None}}
            },
            status=200
        )

        client = VultrClient(api_key="test_key")
        instances = client.get_paginated("/instances")

        assert len(instances) == 3
        assert instances[0]["id"] == "1"
        assert instances[2]["id"] == "3"


class TestVultrBillingCollector:
    """Test billing collector"""

    def test_collect_account_info(self):
        """Test account info collection"""
        mock_client = Mock()
        mock_client.get_account_info.return_value = {
            "account": {
                "name": "test_account",
                "email": "test@example.com",
                "balance": 150.50,
                "pending_charges": 45.30
            }
        }

        collector = VultrBillingCollector(mock_client)
        result = collector.collect_account_info()

        assert result["balance"] == 150.50
        assert result["pending_charges"] == 45.30

    def test_collect_invoices(self):
        """Test invoice collection"""
        from datetime import datetime, timedelta

        mock_client = Mock()
        mock_client.list_invoices.return_value = [
            {
                "id": "inv_1",
                "date": datetime.utcnow().isoformat() + "Z",
                "amount": 100.00
            },
            {
                "id": "inv_2",
                        "date": (datetime.utcnow() -
timedelta(days=100)).isoformat() + "Z",
                        "amount": 200.00
                    }
                ]

                collector = VultrBillingCollector(mock_client)
                invoices = collector.collect_invoices()

                # Should only get recent invoice (within 90 days)
                assert len(invoices) == 1
                assert invoices[0]["invoice_id"] == "inv_1"


        class TestVultrInstanceCollector:
            """Test instance collector"""

            def test_collect_compute_instances(self):
                """Test compute instance collection"""
                mock_client = Mock()
                mock_client.list_instances.return_value = [
                    {
                        "id": "inst_1",
                        "label": "web-server",
                        "plan": "vc2-1c-1gb",
                        "status": "active",
                        "power_status": "running",
                        "monthly_cost": 5.00
                    },
                    {
                        "id": "inst_2",
                        "label": "gpu-server",
                        "plan": "vhf-8c-32gb-gpu",
                        "status": "active",
                        "power_status": "running",
                        "monthly_cost": 90.00
                    }
                ]

                collector = VultrInstanceCollector(mock_client)
                instances = collector.collect_compute_instances()

                assert len(instances) == 2
                assert instances[0]["is_gpu"] is False
                assert instances[1]["is_gpu"] is True

            def test_analyze_instance_utilization(self):
                """Test instance utilization analysis"""
                instances = [
                    {
                        "instance_id": "i1",
                        "status": "active",
                        "power_status": "running",
                        "is_gpu": False,
                        "monthly_cost": 5.00
                    },
                    {
                        "instance_id": "i2",
                        "status": "active",
                        "power_status": "stopped",
                        "is_gpu": True,
                        "monthly_cost": 90.00
                    }
                ]

                collector = VultrInstanceCollector(Mock())
                analysis = collector.analyze_instance_utilization(instances)

                assert analysis["total_instances"] == 2
                assert analysis["gpu_instances"] == 1
                assert analysis["idle_instances"] == 1
                assert analysis["idle_cost"] == 90.00


        class TestVultrCostAnalyzer:
```

```python
            """Test cost analyzer"""

            def test_analyze_costs(self):
                """Test cost analysis"""
                account_info = {"balance": 100}
                pending_charges = {"pending_charges": 50}
                instances = [
                    {
                        "instance_id": "i1",
                        "is_gpu": False,
                        "power_status": "running",
                        "monthly_cost": 5
                    },
                    {
                        "instance_id": "i2",
                        "is_gpu": True,
                        "power_status": "stopped",
                        "monthly_cost": 90
                    }
                ]
                invoices = []

                analyzer = VultrCostAnalyzer()
                analysis = analyzer.analyze_costs(
                    account_info, pending_charges, instances, invoices
                )

                assert analysis["instance_count"] == 2
                assert analysis["waste_analysis"]["idle_instances"] == 1
                assert len(analysis["recommendations"]) >= 1
                assert analysis["total_estimated_savings"] == 90  # Delete
idle GPU


        @pytest.mark.integration
        def test_collect_vultr_metrics_integration():
            """
            Integration test for full collection.
            Requires VULTR_API_KEY environment variable.
            """
            import os

            api_key = os.getenv("VULTR_API_KEY")
            if not api_key:
                pytest.skip("VULTR_API_KEY not set")

            metrics = collect_vultr_metrics(api_key)

            assert "account" in metrics
            assert "instances" in metrics
            assert "cost_analysis" in metrics
            assert "collected_at" in metrics
```

---

## 📁 FILE STRUCTURE

After implementation:

```
services/cost-agent/
├── src/
│   ├── collectors/
│   │   ├── aws/           # Existing
│   │   ├── gcp/           # Existing
│   │   ├── azure/         # Existing
│   │   └── vultr/         # ✅ NEW
│   │       ├── __init__.py
│   │       ├── client.py        # Vultr API client
│   │       ├── billing.py       # Billing collector
│   │       ├── instances.py     # Instance collector
│   │       └── analyzer.py      # Cost analyzer
│   └── ...
├── tests/
│   ├── test_aws_collector.py   # Existing
│   ├── test_gcp_collector.py   # Existing
│   ├── test_azure_collector.py # Existing
│   └── test_vultr_collector.py # ✅ NEW
└── requirements.txt             # ✅ UPDATED
```

---

## 🔑 KEY CONCEPTS

### 1. API Authentication

Vultr uses Bearer token authentication:

```
headers = {
    "Authorization": f"Bearer {api_key}"
}
```

### 2. Rate Limiting

Vultr limits to 30 calls per second. We implement conservative 500ms delays:

```
time.sleep(0.5)  # Wait between calls
```

### 3. Pagination

Vultr uses cursor-based pagination:

```
{
    "data": [...],
    "meta": {
        "links": {
            "next": "cursor_string"
        }
    }
}
```

### 4. Billing Periods

Different billing models: VMs use 672-hour months, GPUs use 730-hour months

### 5. Cost Attribution

Costs are attributed by: - **Product type** (Compute, GPU, Storage, etc.) - **Region** - **Instance tags** - **Time period**

---

## 🎯 NEXT STEPS

After completing this phase:

1. **Verify:** Run `PHASE1-1.41_PART2_Execution_and_Validation.md`
2. **Integration:** Connect to Cost Agent API endpoints
3. **Testing:** Run with real Vultr account
4. **Optimization:** Implement cost-saving recommendations

---

## 📚 REFERENCES

### Vultr Documentation

- **API Docs:** https://www.vultr.com/api/
- **Billing API:** https://www.vultr.com/api/#tag/billing
- **Pricing:** https://www.vultr.com/pricing/

### Python Libraries

- **requests:** HTTP client
- **aiohttp:** Async HTTP client
- **tenacity:** Retry logic

---

**Document Version:** 1.0
**Last Updated:** October 21, 2025
**Status:** ✅ Ready for Implementation